NPSCS-92-007

NAVAL POSTGRADUATE SCHOOL
Monterey, California

SYSTEMATIC DEVELOPMENT OF HARD REAL-TIME SOFTWARE: A COMPARATIVE STUDY OF THREE METHODS

by
Yuh-jeng Lee
Luqi
Valdis Berzins

April 1992

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAR ADMIRAL R. W. WEST, JR.
Superintendent

HARRISON SHULL
Provost

This report was prepared for and funded by the Naval Postgraduate School.

This report was prepared by:

VALDIS BERZINS
Associate Chairman for
Technical Research

PAUL MARTO
Dean of Research

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-92-007	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Naval Postgraduate School Monterey, CA 93943-5100
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER OM & N Direct Funding
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Sytematic Development of Hard Real-Time Software: A Comparative Study of Three Methods

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	April 1992	32

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Hard real-time systems, Systematic Software development, Structured analysis, Computer aided prototyping, the Spec language.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present a comparative study on three software development methods which cover the entire development life cycle for hard real-time systems: (1) Structured Analysis, (2) Computer Aided Prototyping, and (3) Spec formal logic specification method. We use a simple example to demonstrate the software development process using all three approaches. The strengths and weaknesses of each method are discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Yuh-jeng Lee	22b. TELEPHONE (Include Area Code) (408) 646-2361 22c. OFFICE SYMBOL CS/Le

Systematic Development of Hard Real-Time Software: A Comparative Study of Three Methods

Yuh-jeng Lee
Luqi
Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey CA 93943

Abstract

We present a comparative study on three software development methods which cover the entire development life cycle for hard real-time systems: (1) Structured Analysis, (2) Computer Aided Prototyping, and (3) Spec formal logic specification method. We use a simple example to demonstrate the software development process using all three approaches. The strengths and weaknesses of each method are discussed.

Keywords: Hard real-time systems, Systematic software development, Structured analysis, Computer aided prototyping, the Spec language.

1 Introduction

Real-time software has to execute multiple processes efficiently according to timing constraints, and to provide mechanisms for synchronous and asynchronous process communication. It is usually embedded in large, complex systems such as flight control systems. The correctness of such software depends on the time at which the results are produced as well as the logical results of computation. Although real-time system design is mostly ad hoc in current practice, there have been attempts to make this process a scientific endeavour [Mok-88]. Many problems such as specification and verification techniques, process scheduling, communication architectures, and automated/systematic hard real-time software development methods that cover the entire software life cycle, remain to be solved [Stankovic-88].

Typically, a real-time system includes a set of independent hardware devices that operate at widely differing speeds. Real-time systems can fail because the system is unable to execute its critical workload in time. It is important to utilize the available finite resources within the system intelligently and with predictable results. A method for developing real-time systems, therefore, needs to provide facilities to solve the following problems: meeting stringent time requirements and performance specifications; processing and protection of messages that arrive at irregular intervals, with variable input rates and different priorities; mapping concurrent demands into a proper set of concurrent processes; allocating concurrent processes to multiple processors, if more than one is available; handling synchronization between concurrent processes for intra- as well as

inter-processor communication and protecting shared data: controlling hardware devices such as communication lines, sensors, actuators, and computer resources; designing software simulators for the hardware devices that are not available for the test phase; and testing and debugging the system.

Methods for developing real-time systems must also address problems common to developing any large software system, including the size of the development team, complexity of the system, cost of communication between team members, reliability of the system, and the life cycle maintenance of the system. A method for developing real-time systems, therefore, also needs to address and support systematic planning and management, simple and manageable conceptual models, a modular design, precise design documents (on-line), configuration control, extreme programming accuracy, and localized decisions in single modules.

1.1 The Target System

Requirements of the Fish Farm Control System (FFCS): We are going to develop a FFCS which is a small example of a typical embedded real-time control system. The FFCS will control the fish food dispenser and water quality in a fish pond. The pond has a mechanical feeder that drops pellets of fish food from a feeder tube suspended above the pond. The feeder can be turned on and off by the computer. The pond also has a water inlet pipe and a drain pipe with valves controlled by the computer, and sensors that measure the water level (millimeters above the bottom), the oxygen level in the water (parts per million), and the ammonia level in the water (parts per million).

The FFCS must deliver fish food at scheduled feeding times, repeated every day. The times when each feeding starts and stops are displayed on the console of the FFCS and can be adjusted from the keyboard.

The FFCS must keep the water level between 500 and 1000 mm, the oxygen level at least 6 ppm, and the ammonia level at most 10 ppm. If the inlet valve setting is greater than the drain valve setting, then the water level in the pond increases. The water in the inlet pipe is high in oxygen and low in ammonia, so that increasing water flow increases the oxygen level, and decreases the ammonia level. The FFCS must minimize water flow subject to the above constraints.

Hardware Interfaces: There are three separate hardware devices in the FFCS: sensors, valves, and the feeder. As part of the design process we must create the necessary software components that accept inputs from these devices.

1.2 The Development Methods

We have chosen three development methods for our study: (1) Structured Analysis, (2) Computer Aided Prototyping, and (3) Spec formal logic specification method.

The Structured Analysis approach [cf., Yourdon-89] represents a class of informal structured approaches to requirements analysis and system specification. This method is well-known and is representative of state-of-the-art practice in industry. There are a

number of textbooks and supporting software tools. Structured analysis uses three models: the essential model, the user implementation model, and the design model. It applies a top-down stepwise refinement process to capture what the system must do to satisfy the user's requirements in an essential model. It captures information about the automation boundary, user interface, manual support activities, choices of hardware/software technology and the system's operational constraints in a user implementation model. It describes how the system fulfill essential and implementation requirements in a design model which is ready to be built and tested.

The computer-aided prototyping method uses a series of prototypes and feedback from users to converge to a viable software design. Computer Aided Prototyping System (CAPS) [Luqi-Berzins-88] includes an integrated set of CASE tools for rapid prototyping of hard real-time systems, making it possible for prototypes to be designed expeditiously and to be executed for validating the requirements. CAPS uses the Prototyping Description Language (PSDL) to provide module specifications and their interconnections. It also provides automated methods for retrieving, adapting, and combining reusable components based on normalized module specifications; establishing feasibility of real-time constraints via scheduling algorithms; simulating unavailable components via algebraic specifications; automatically generating translators and real-time schedules for supporting execution; constructing a prototyping project database using derived mathematical models; providing automated design completion and error checking facilities in a designer interface containing graphical interface for design and debugging.

The Spec formal specification approach supports software system development via black-box interface specifications [Berzins-Luqi-91]. The Spec language combines logic with a simple underlying event model. It is sufficiently powerful for specifying many kinds of software systems, and sufficiently flexible to allow software designers to express their thoughts without forcing them into a restrictive framework. The Spec language is also formal enough to support computer-aided design of software. Tools currently under investigation include syntax-directed editors, consistency checkers, design completion tools, test case generators and prototype generators.

1.3 Organization of the Paper

Section 2 analyzes the FFCS problem using the Structured Analysis approach to developing real-time software systems. Section 3 uses a semi-automated method supported by CAPS to solve the FFCS problem. Section 4 uses the Spec formal development method for the FFCS problem. Comparisons and concluding comments are in section 5.

2 Structured Analysis

Structured Analysis (SA) [Yourdon-89] is a widely used, informal method. It consists an essential model, a user implementation model, and a design model, followed by implementation, testing, and maintenance activities.

2.1 The Essential Model

The essential model describes what the system must do to satisfy the user's requirements. It has two major components: environmental model and behavioral model. The environmental model defines the boundary between the system and the rest of the world. The behavioral model describes the required behavior of the system.

2.1.1 The Environmental Model

The environmental model for the FFCS consists of the following:

Statement of purpose (purpose of the system):
The purpose of the Fish Farm Control System is to control water conditions and feeding.

The event list (stimuli recognized by the system):
The event list of the FFCS consists of five events: (1) sensor sends oxygen level; (2) sensor sends ammonia level: (3) sensor sends water level: (4) operator changes feeding times; and (5) a feeding time occurs

The context diagram (system boundaries and interactions):
The context diagram, shown in Fig. 1, uses the following symbols:

- **Terminators,** □, represent people, organizations, or other systems with which the system communicates.
- **Data Flows,** —>, and **Control Flows,** - - ->, represent the data that crosses the boundary of the system.
- **Data Stores,** =, represent persistent data shared by the system and the terminators.
- **Boundary,** ◯, represents the boundary between the system and the rest of the world.

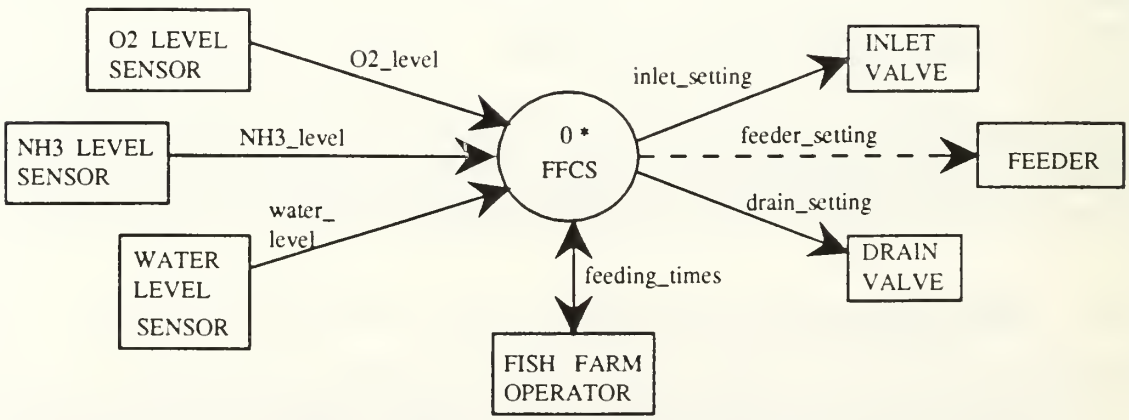


Fig. 1 Context Diagram for the FFCS

2.1.2 The Behavioral Model

The behavioral model for the FFCS consists of the following:

Data Flow Diagram (DFD – system decomposition):

The top level DFD for the FFCS is shown in Fig. 2. In this case the system decomposes into two completely independent subsystems. The data store shows the requirement that the FFCS must store the feeding schedule, so that operator input is needed only to change feeding policies.

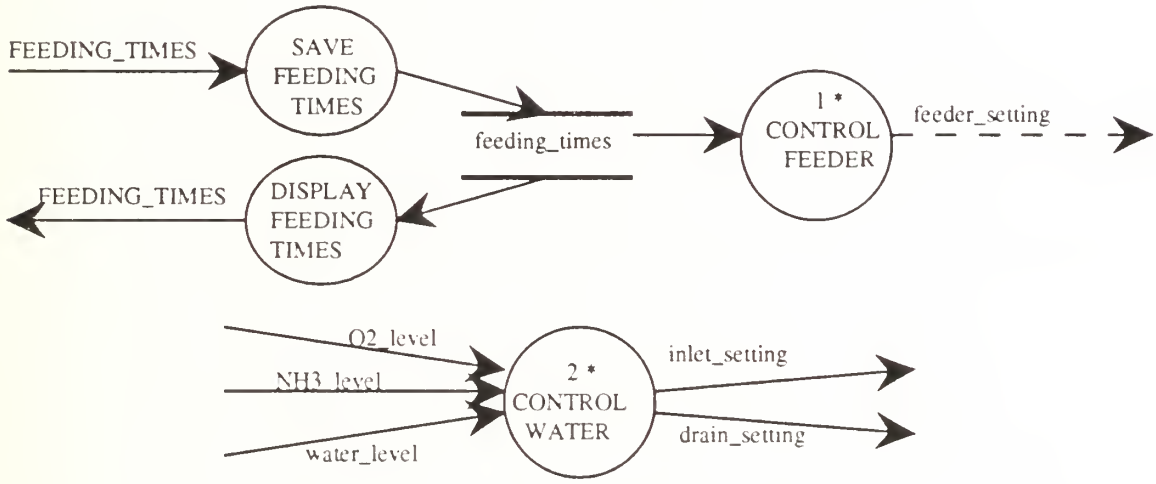


Fig. 2 Top Level Data Flow Diagram

A complex system is explained by decomposing it into simpler subsystems. For example, the bubble CONTROL_WATER in Fig. 2 can be expanded into the diagram as shown in Fig. 3.

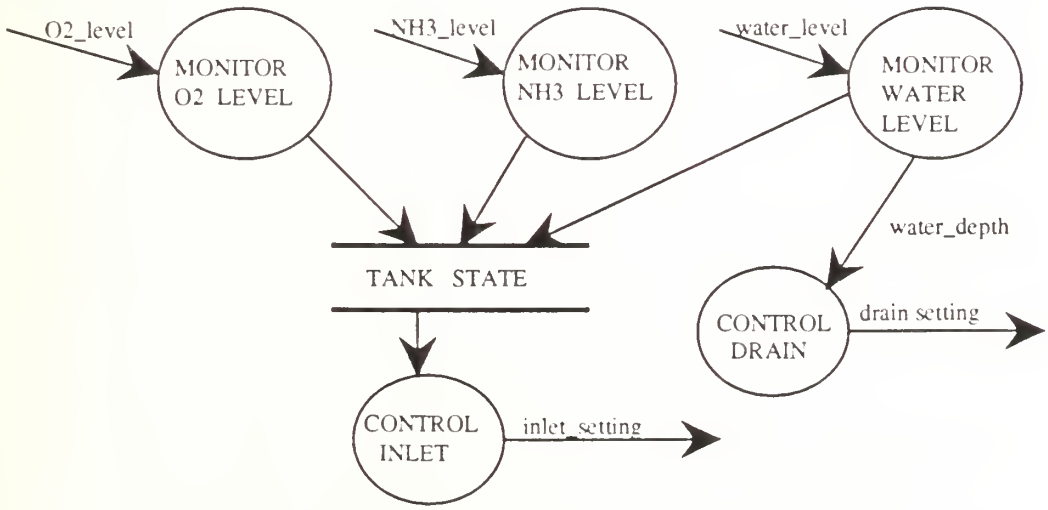


Fig. 3 DFD for CONTROL WATER

The purpose of the data store TANK_STATE is to combine inputs from three asynchronous sensors.

Data Dictionary (system data types):

The data dictionary for the FFCS which defines the data types associated with flows and stores is shown in Fig. 4. It describes the content and interpretation of the data elements in the system.

```
drain_setting   = [ OPEN | CLOSED ]
feeder_setting  = [ ON | OFF ]
FEEDING_TIMES   = start_time + stop_time
feeding_times   = { FEEDING_TIMES }
hours           = * range : 0..23 *
inlet_setting   = [ OPEN | CLOSED ]
minutes         = * range : 0..59 *
NH3_level       = * ammonia level in water (parts per million) *
NH3_quality     = [ HIGH | OK ]
NOW             = * the current time of day *
O2_level        = * oxygen level in water (parts per million) *
O2_quality      = [ LOW | OK ]
start_time      = hours + minutes * time of day *
stop_time       = hours + minutes * time of day *
TANK_STATE      = water_depth + NH3_quality + O2_quality
water_depth     = [ LOW | OK | HIGH ]
water_level     = * water level in the pond (millimeters) *
```

Fig. 4 Data Dictionary

Process Specifications: The process specifications for the FFCS are shown in Fig. 5.

```
MONITOR_O2:
   IF O2_level < 6 ppm THEN set O2_quality = LOW
                       ELSE set O2_quality = OK
MONITOR_NH3:
   IF NH3_level > 10 ppm THEN set NH3_quality = HIGH
                         ELSE set NH3_quality = OK
MONITOR_WATER:
   IF water_level < 50 cm THEN set water_depth = LOW
   ELSE IF water_level > 100 cm THEN set water_depth = HIGH
   ELSE set water_depth = OK
CONTROL DRAIN:
   IF water_depth = HIGH THEN set DRAIN_SETTING = OPEN
                         ELSE set DRAIN_SETTING = CLOSED
CONTROL INLET:
   IF water_depth = LOW OR O2_quality = LOW OR NH3_quality = HIGH
      THEN set inlet_setting = OPEN ELSE set inlet_setting = CLOSED
```

Fig. 5 Process Specifications

The process specifications are informal explanations of the required behavior of bubbles that are not decomposed. The example shows the simplest plausible control policy for water quality.

Entity Relationship Diagram (ERD – database schema):

An ERD consists of the following components:

- An *entity*, □, represents a collection of objects.
- A *relation*, ◇, represents a set of connections between the entities.
- An *attribute* is used to describe a property of an entity.

The ERD for the FFCS is shown in Fig. 6. The diagram is very simple because there is only one data store containing a set of objects, and there are no stored relationships between data stores in this case.

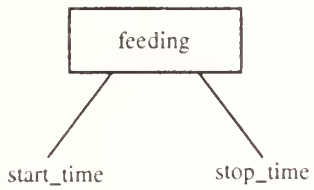


Fig. 6 Entity Relationship Diagram

State Transition Diagram (STD – time dependent control policies):

The STD for the CONTROL_FEEDER subsystem of the FFCS, shown in Fig. 7, consists of the following components:

- States, □ , represent states of a finite state machine.
- Transitions, ⟶, represent possible state changes.
- Annotations that specify the conditions under which transition must occur and the actions associated with each transition, in the form $\frac{condition}{action}$.

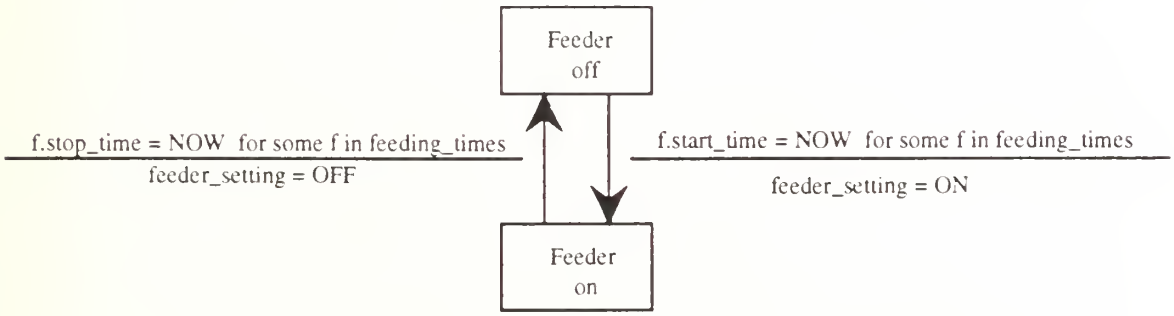


Fig. 7 State Transition Diagram for CONTROL FEEDER

In this case the conditions depend both on the current value of a real-time clock and the current version of the stored feeding schedule.

2.2 The User Implementation Model

The user implementation model for FFCS consists of the following:

Technology to be used: personal computer with minimum 512K RAM, a standard keyboard and monitor, three hardware input connections for sensors, three hardware output connections for valves and feeder, and operating system that supports compiled Ada code.

Automation boundary: All processes within the essential model are considered within automation boundary.

User interface: interactive, allowing entry of feeding time, displaying feeding times, and for testing purposes, displaying valve settings and sensor readings.

Coding requirements: Ada language using top-down and structured programming .

Manual support activities: Loading the fish feeder.

Operating constraints: Maximum of 10 feeding times allowed per day,

Reliability requirements: Time to repair less than one hour is required to maintain stock of fish. Redundant systems with a back up power source are recommended.

2.3 The Design Model

The design model for the FFCS consists of the following:

The Processor Model (allocation of processes to processors): The entire essential model of FFCS is allocated to a single processor.

The Task Model (identification of tasks on each processor): Three tasks identified for FFCS are: (1) controlling water flow, (2) displaying and adjusting feeding times, and (3) controlling the feeder.

The relations among the three tasks are shown in Fig.8.

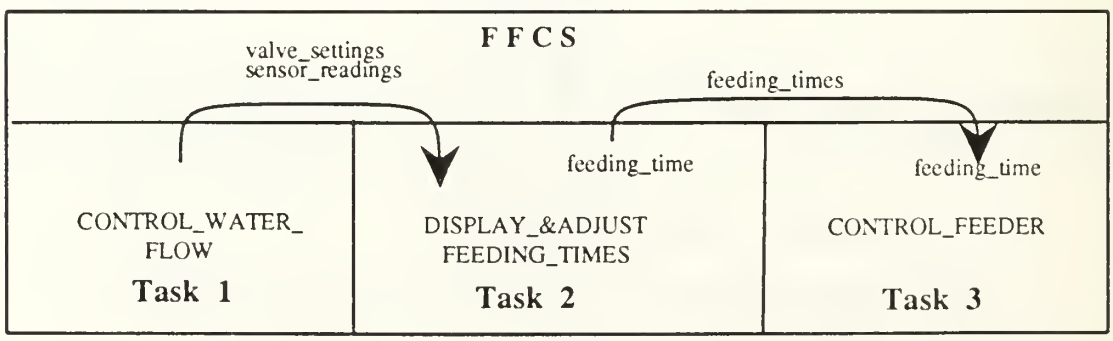
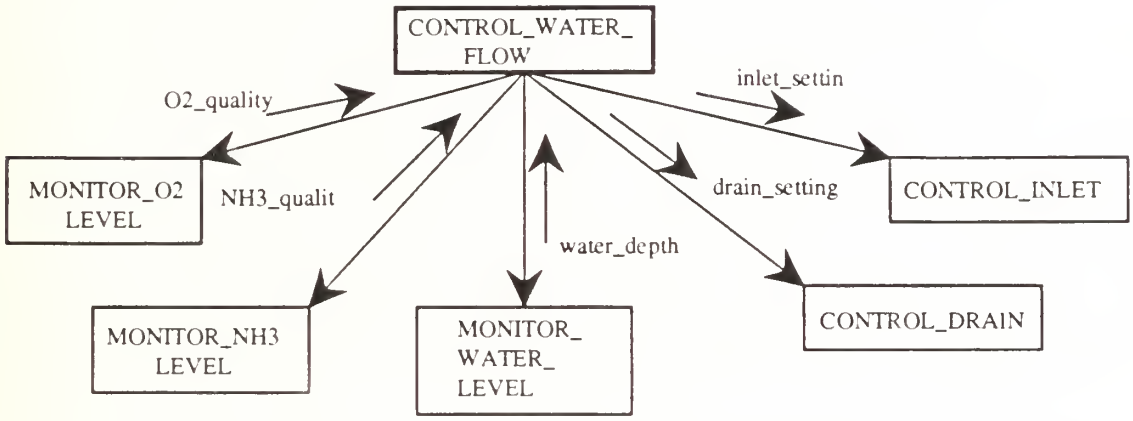


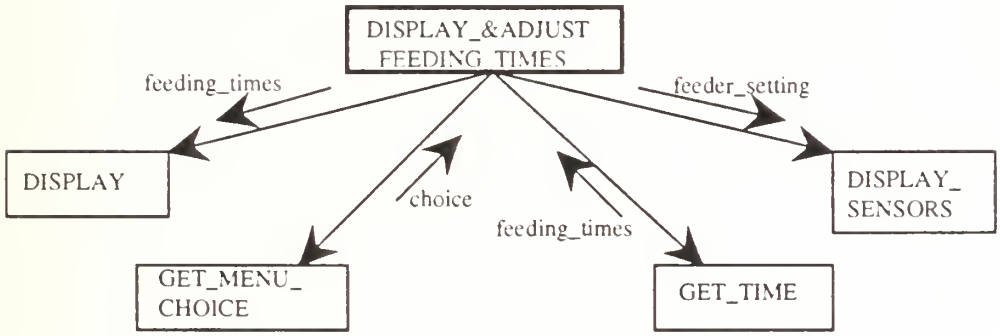
Fig. 8 Relationships between Tasks

The Program Model (hierarchical module structure for each task): The structure charts for the three tasks in FFCS are shown in Fig. 9 – 11. The program model transforms asynchronous processes in the DFD into synchronous subprograms to improve efficiency, based on the designer's judgement.



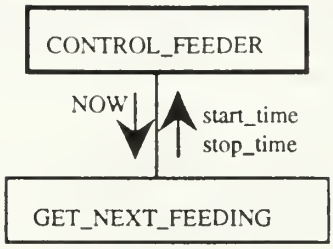
MONITOR_O2_LEVEL -- measures the oxygen level in water and stores it.
MONITOR_NH3_LEVEL -- measures the ammonia level in water and stores it.
MONITOR_WATER_LEVEL -- measures the water lever and store it.
CONTROL_DRAIN -- sets the drain valve due the the sensor readings when activated.
CONTROL_INLET -- sets the inlet valve due to the sensor readings when activated.

Fig. 9 Structure chart for Task 1



GET_TIME -- gets the new feeding schedule from the user.
GET_MENU_CHOICE -- displays a menu and gets the user choice.
DISPLAY -- displays all feeding times in the data store.
DISPLAY_SENSORS -- displays current sensor readings and valve settings.

Fig. 10 Structure chart for Task 2



GET_NEXT_FEEDING -- gets the end points of the next feeding time

Fig. 11 Structure chart for Task 3

Typical design goals such as high cohesion within modules, low coupling between modules, and small module size (one page of code) serve as guidelines during the process. There is no sure way to enforce these guidelines in structured analysis.

2.4 Programming and Testing

This step includes (1) choosing program units in a programming language for the modules in the design, (2) deriving test cases from the specification of the response to each event covered by the design, and (3) describing the input data and expected output for the test case to confirm to the specification in user implementation model. These are all manual activities.

The FFCS is implemented using the Ada programming language. The Ada interface specification to the FFCS hardware is given below:

```
package FFCS_hardware_interface is
  type valve_setting is (open, closed);
  type feeder_setting is (on, off);

  function water_level return natural;  -- Units of mm.
  function oxygen_level return natural;  -- Units of ppm.
  function ammonia_level return natural;  -- Units of ppm.

  procedure set_feeder(s: feeder_setting);
  procedure set_inlet_valve(s: valve_setting);
  procedure set_drain_valve(s: valve_setting);
end FFCS_hardware_interface;
```

2.5 Maintenance

Software development often requires substantial maintenance effort due to changes in the original requirements. Requirements changes could cause changes in all of the documents shown in Sections **2.1 – 2.4**. Such a costly operation is only weakly supported by existing tools such as Software Through Pictures [STP-90], considering the tasks need to be accomplished.

3 Computer Aided Prototyping

The Computer Aided Prototyping System, CAPS, has been designed for hard real-time system development and iterative adjustment of requirements. The CAPS method is based on an iterative process for discovering viable system structures. Through rapid construction of executable prototypes, the behaviors of the expected system can be demonstrated to the user for early feedback. To decide whether a system being built really fulfills the user's need, instead of having the user inspecting formal requirements documents in the beginning stage or waiting until the end of the project to observe the system, CAPS offers real-time execution support to demonstrate system behavior before actual implementation takes place. In addition to meeting diverse needs for both the user and the designer, CAPS also contains software tools for automatic code generation and incorporating reusable components to produce executable prototypes with real-time schedules. The method and the supporting tool are described in [Luqi-91].

3.1 Iterative Prototyping Process

The process of developing software using CAPS consists of (1) determining initial requirements (2) constructing the prototype; (3) demonstrating prototype behavior; (4) adjusting requirements and iterating; and (5) when stable, implementing and optimizing the system.

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g., in English) or using a formal notation. These requirements may be refined by asking the user to verify their completeness and correctness.

The next step is to use PSDL (Prototype System Description Language) for the specification of the prototype, according to the requirements. It involves the construction of dataflow diagrams enhanced with nonprocedural timing and control constraints, data stream types, and unified data and control flows. A PSDL description of a system or subsystem contains a specification part and an implementation part. The specification part is used to determine whether the required functionalities of a system or subsystem (called an operator in CAPS) can be performed by a reusable component already in the software base. If this is indeed the case, the implementation part is simply an Ada module (or whatever language the component is written in) and the component is called *atomic*. Otherwise, the component is *composite* and needs to be decomposed into more primitive operations if possible, or to be hand-coded. This process is illustrated in Fig. 12.

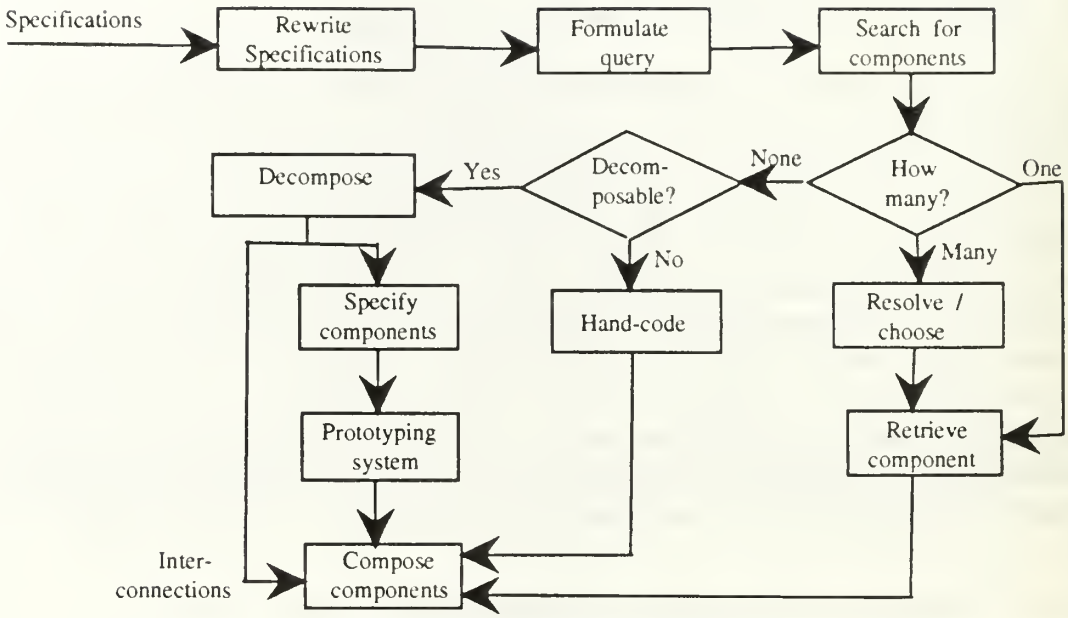


Fig. 12 Prototyping in CAPS

When all the components are integrated into a prototype, the execution support of CAPS provides an efficient way to demonstrate prototype behavior. The translator binds all the components together and generates executable Ada code for the prototype. The static scheduler analyzes all hard real-time constraints to construct a schedule that guarantees all time-critical computations meet their requirements. The dynamic scheduler schedules the computations that do not have hard real-time constraints in time slots not used by the time-critical computations. The debugger exercises the prototype and allows modifications to be made. Changes can be done at the PSDL specification level and a revised prototype can be assembled for further demonstration. This process can be repeated until a satisfactory prototype is produced, as illustrated in Fig. 13.

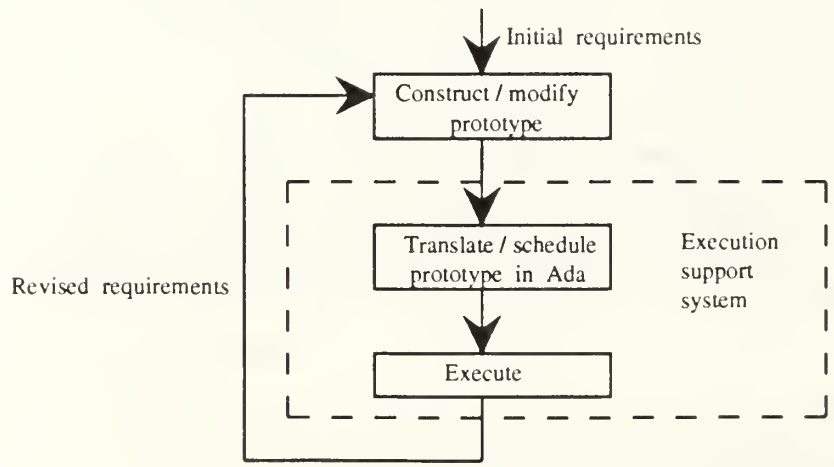


Fig. 13 Iterative Prototyping Process in CAPS

The process of software development using CAPS consists of the following steps:

1. Determine initial requirements
2. Construct prototype
   - Find reusable components
   - Decompose
   - Write Ada code
3. Demonstrate
   - Generate schedule and executable
   - Demo typical scenarios
   - Get feedback
4. Adjust requirements and iterate
5. When stable, implement and optimize
   - Complete non-critical parts
   - Transform to gain efficiency
   - Port to operating environment

3.2 Initial Requirements for the FFCS

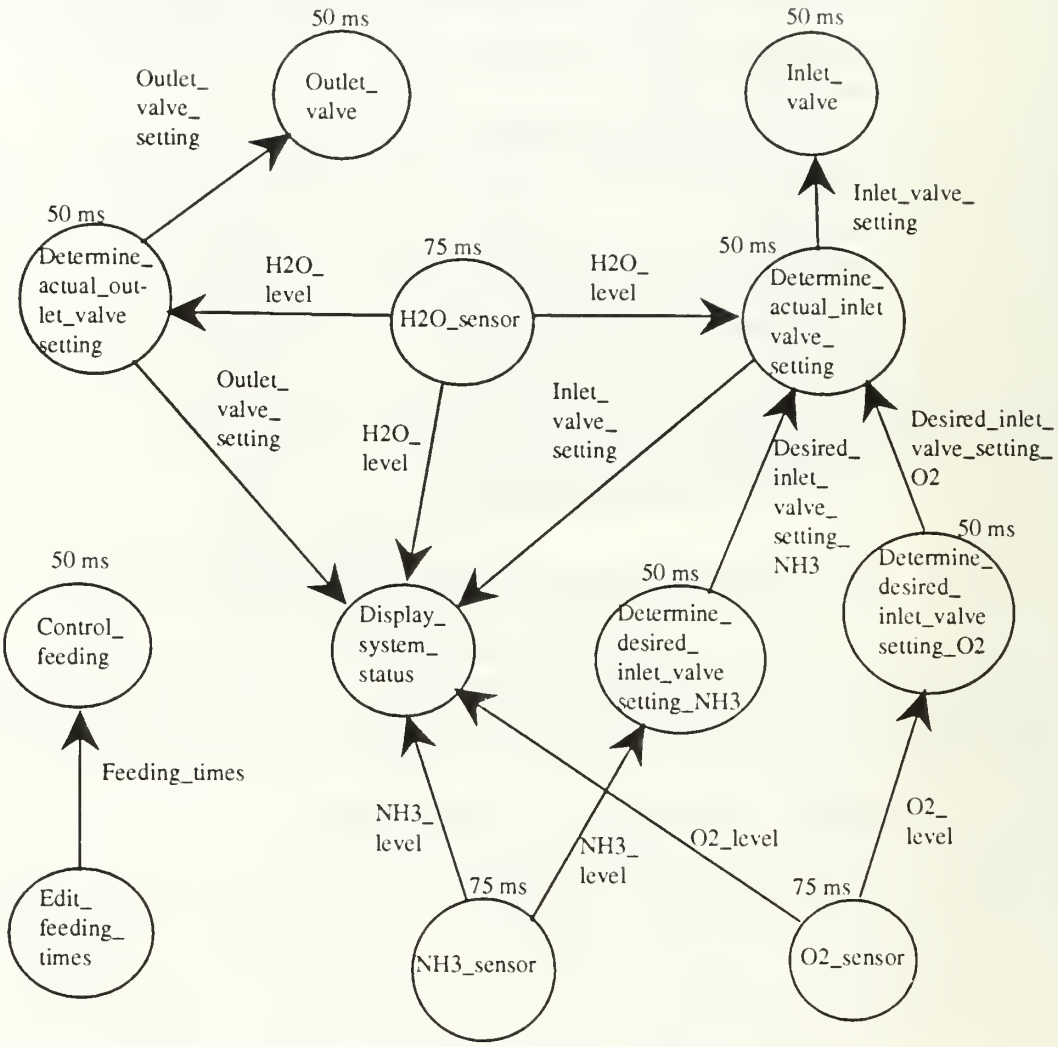
The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. The FFCS Requirements presented in Section 1.1 are used for the construction of the FFCS software prototype.

3.3 FFCS Prototype Construction

Since a reusable FFCS is not available, we decompose the system into a set of operators. Refinements can be made in PSDL by (1) adding constraints to the specifications and retrieving new reusable components or (2) doing further decompositions to make the implementation correspond to the refined specification. Optimizations are performed at the PSDL level by introducing alternative decompositions that eliminate unnecessary processing or allow more efficient algorithms. This results in a preliminary design free from programming level details.

The PSDL description for the FFCS is given in Fig.14.

```
OPERATOR fishfarm
     SPECIFICATION
          DESCRIPTION { Prototype of the fish farm control system
                        and the systems it controls }
          STATES feeding_times: set[feeding_time] INITIALLY empty
     END
     IMPLEMENTATION
          GRAPH
```




```
DATA STREAM
     Inlet_valve_setting, Outlet_valve_setting,
     Desired_inlet_valve_setting_O2,
     Desired_inlet_valve_setting_NH3 : valve_setting,
     H2O_level, NH3_level, O2_level : natural,
CONTROL CONSTRAINTS
     OPERATOR Outlet_valve MAXIMUM RESPONSE TIME 1 sec
     OPERATOR Determine_actual_outlet_valve_setting
         MAXIMUM RESPONSE TIME 1 sec
     OPERATOR H2O_sensor MAXIMUM RESPONSE TIME 1 sec
     OPERATOR Control_feeding MAXIMUM RESPONSE TIME 1 sec
     OPERATOR Edit_feeding_times MAXIMUM RESPONSE TIME 500 ms
     OPERATOR Display_system_status MAXIMUM RESPONSE TIME 500 ms
     OPERATOR NH3_sensor MAXIMUM RESPONSE TIME 1 sec
     OPERATOR Determine_desired_inlet_valve_setting_NH3
         MAXIMUM RESPONSE TIME 1 sec
     OPERATOR Determine_actual_inlet_valve_setting
```

```
        MAXIMUM RESPONSE TIME 1 sec
    OPERATOR Inlet_valve MAXIMUM RESPONSE TIME 1 sec
    OPERATOR Determine_desired_inlet_valve_setting_O2
        MAXIMUM RESPONSE TIME 1 sec
    OPERATOR O2_sensor MAXIMUM RESPONSE TIME 1 sec
END
```

Fig. 14 PSDL Description of the FFCS

According to the CAPS method, the FFCS operator can be decomposed into the network of operators as shown in the graph of Fig. 14. This graph is different from a DFD in structured analysis in several ways: (1) each operator in the graph has an associated maximum execution time for that operator – this is a constraint that must be satisfied by the scheduler; (2) each operator can have annotated control constraints specifying conditional requirements for the firing of operators; (3) data streams are implicitly buffered so that data stores are not necessary; (4) hardware components such as sensors are included in the representation – this not only gives a complete picture of the source and destination of a data stream, but also provides important information for interface design; (5) formal descriptions of the representation are generated automatically, in PSDL (Prototype System Description Language), which allows for schedulability analysis as well as code synthesis.

3.4 FFCS Prototype Demonstration

The CAPS execution support system includes a translator, static scheduler, and dynamic scheduler. The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints. The debugger lets the designer control and examine the execution of the prototype.

4 The Spec Method

The Spec language [Berzins-Luqi-91] is a formal specification language designed to support conceptual modeling and helps analysts impose structure on apparent chaos by factoring out coherent pieces in an active process that combines discovery with reformulation, concept formation, generalization, and reorganization. It can specify the behavior of three different types of software modules:

1. Functions: do not exhibit internal memory; output depends only on current input.
2. Machines: modules with internal memory; output depends on both internal state and input.

3. Types: abstract data types provides a set of instances and a set of operations involving the instances. There are two kinds of types: (1) immutable types, whose instances are *values* without internal states (for example, numbers); and (2) mutable types, whose instances are *objects* with internal states (for example, windows).

These modules can interact only via three different types of messages: normal messages, exceptions, and generators.

4.1 FFCS Requirements Analysis using Spec

Spec is logic based and provides a formal mechanism to record and validate requirements which should be precise, testable, and feasible. To achieve this, analysts must formalize the initial problem statement so that it is subject to mathematical modeling. This also aids discovery of unstated requirements and resolution of inconsistencies. Analysts must determine constraints on the development, and analyze, design, or implement high-risk parts to establish feasibility. The use of logic in Spec allows one to formally define laws expressing or partially describing the semantics of new attributes and relationships. Fig. 15 illustrates the Spec requirements formulation method.

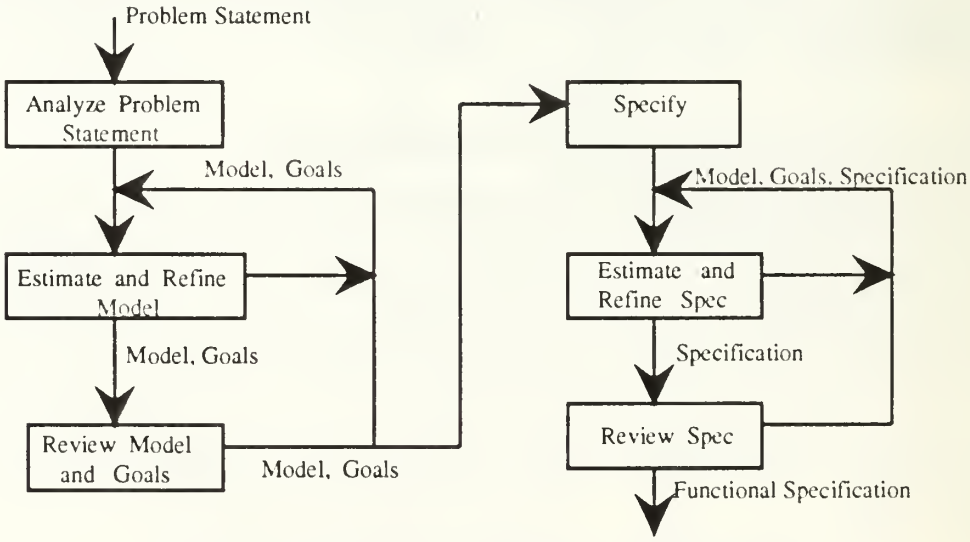


Fig. 15 Requirements Formulation in Spec

A requirements document contains the following:

1. *An environment model*: This is the basis for communication and agreement between the customer and the developer. The model defines the concepts needed for describing the world in which the proposed system will operate.

2. *Goals*: These define the automation boundary of the system and document the relation of software requirements to general system requirements and design goals.

3. *Constraints*: This include implementation constraints (hardware, operating system, and programming language), performance constraints (time and space), and resource constraints (budget and schedule).

Given these components, the analysts propose external interfaces for the system meeting the formalized requirements. This is the process of functional specification – deciding *what* is to be built, not how to build it. This completes the requirements analysis.

4.1.1 Deriving Goals and Subgoals

Following the procedure describe above, we proceed as follows:

Statement of purpose: The purpose of the Fish Farm Control System is to control the fish feeding and water quality (oxygen, ammonia, water level) in the fish pond. This is accomplished by monitoring water quality and adjusting water flow. The feeding times shall be controlled by the fish farm operator.

High level goals and requirements: These are derived from the initial problem statement and are usually organized into a hierarchy, with lower level goals specifying in more detail how the system is supposed to realize the higher level goals, as shown in Fig. 16.

```
G1: The goal of the fish farm is to make a profit by selling fish.
    The fish farm would like to maximize it profits.
  G1.1 The fish farm would like to maximize its production of fish.
    R1.1.1 The fish farm control system must provide a sufficient
           supply of fish food.
    R1.1.2 The fish farm control system must provide adequate
           water quality.

  G1.2 The fish farm would like to minimize its costs.
    R1.2.1 The fish farm control system must avoid waste of fish food.
    R1.2.2 The fish farm control system must avoid excess water flow.
    R1.2.3 The fish farm control system must decrease labor costs.
```

Fig. 16 Top Level Goals

Requirements `R1.1.1` can further be refined, by locating the undefined concepts in the high-level goals, and determining in more detail what they are supposed to mean, as illustrated in Fig. 17.

```
R1.1.1 The fish farm control system must provide
       a sufficient supply of fish food.
   G1.1.1.1 The operator must decide the feeding schedule for fish food.
            The feeding schedule defines how much food
            should be delivered at what time.
     R1.1.1.1.1 The fish farm control system must provide
                a means to display the current feeding schedule.
```

```
                See event display_feeding_schedule.
      R1.1.1.1.2 The fish farm control system must provide
                a means to update the current feeding schedule.
                See event update_feeding_schedule.
      G1.1.1.2 The operator is responsible for keeping an adequate
               supply of fish food in the fish feeder.
      R1.1.1.3 The fish farm control system must automatically deliver
               fish food according to the schedule given by the operator.
               See definition fish_fed_on_time.
               See temporal event feeding_time.
```

Fig. 17 Refinement of R1.1.1

R1.1.2 and G1.2 can be refined in a similar way, to reach goals that can be mapped into the set of stimuli to which the FFCS must respond.

This hierarchy identifies stimuli recognized by the FFCS, shows the justifications for the software requirements in terms of the goals of the entire system and the customer organization, and is the basis for negotiation of the automation boundary. This document records and makes visible decisions that are implicit in the other methods, so that they cna be reviewed at an early stage. The goal hierarchy is also useful for training the impact of changes in system requirements on the software requirements, particularly for very large systems.

4.1.2 Formalizing Goals and the Environment Model

The goals in the hierarchy are usually informal and very general at the top, and get more specific at the lower levels. It is sometimes useful to formalize the goals at the leaves, especially if the system concepts are unfamiliar to the developers or if the code is to be mathematically proven to meet the requirements. The FFCS example is relatively simple and easy to understand informally, so that formalization of the goals and domain concepts is not really necessary. To illustrate how the method would be used in more complex applications, we show the formalized goals for the FFCS in Fig. 18.

```
DEFINITION fish_farm_control_system_requirements
  INHERIT feeding_schedule_definitions
  INHERIT fish_farm_control_system_definitions -- context diagram information
  INHERIT water_quality_definitions -- gives values for symbolic constants
  IMPORT goal FROM requirement_model -- reusable concepts
  IMPORT now FROM time_definitions -- reusable concepts

  CONCEPT fish_fed_on_time: boolean -- R1.1.1.3
    WHERE fish_fed_on_time <=>
      (fish_feeder_on <=> feeding_time(current_feeding_cschedule, now)),
      goal(fish_fed_on_time, fish_farm_control_system)

  CONCEPT safe_oxygen_level: boolean -- R1.1.2.1
    WHERE safe_oxygen_level <=> oxygen_level >= minimum_oxygen_level,
      goal(safe_oxygen_level, fish_farm_control_system)
```

```
  CONCEPT safe_ammonia_level: boolean -- R1.1.2.2
    WHERE safe_ammonia_level <=> ammonia_level <= maximum_ammonia_level ,
      goal(safe_ammonia_level, fish_farm_control_system)

  CONCEPT safe_water_level: boolean -- R1.1.2.3
    WHERE safe_water_level <=>
            minimum_water_level <= water_level <= maximum_water_level,
      goal(safe_water_level, fish_farm_control_system)
END
```

Fig. 18 Formalized Goals

These goals are expressed in the Spec language [Berzins-Luqi-91]. Specifications in the Spec language are organized in units called *modules*. The modules used in requirements analysis are delimited by the keywords DEFINITION and END, and contain definitions for a set of *concepts* which represent objects and properties of the problem domain and are used to describe software systems. INHERIT and IMPORT support controlled sharing of concepts without replicating definitions. WHERE is followed by logical assertions that describe the meanings of the concepts precisely but without forcing overspecification of details. Concepts can also be defined informally using English comments.

The environmental model is the result of a domain analysis. Such a domain analysis is optional, but it can be useful if many systems that address similar problems are to be developed, because the environment model becomes the organizing structure for reuse and sharing of designs and code. The environmental model is built by identifying and defining the concepts needed to formalize the goals. For example, goal R1.1.1.3 introduces the concept of a feeding schedule, which is defined in Fig. 19. In more complex examples, constructing a formal environment model can help identify implicit requirements, which are added to the goal hierarchy.

```
DEFINITION feeding_schedule_definitions
  IMPORT time FROM time_definitions

  CONCEPT feeding_schedule: type
  CONCEPT current_feeding_schedule: feeding_schedule
  CONCEPT feeding_time(f: feeding_schedule, t: time)
    VALUE(b: boolean)
  CONCEPT time_to_start_feeding(f: feeding_schedule, t: time)
    VALUE(b: boolean)
    WHERE b <=> feeding_time(f, t) &
                SOME(t1: time ::
                 ALL(t2: time :: t1 <= t2 < t => ~feeding_time(f, t2)))
END
```

Fig. 19 Properties of Feeding Schedules

4.1.3 Constraints

Three kinds of constraints are relevant to requirements analysis:

1. implementation constraints: hardware, operating system, and programming language;
2. performance constraints: time, space, and reliability;
3. resource constraints: budget and development schedule.

Many of the implementation constraints usually come from the initial problem statement and are straightforward to specify. Performance constraints are sometimes derivable from the characteristics of the problem domain and can be either flexible or absolute. Typically, limits on resources are given by the customer with relatively little flexibility, and the analyst must specify the best system that can be achieved within those limits. The constraints for the fish farm control system are given as follows:

```
C1: The fish farm control system must be implemented in Ada.
C2: The project must be completed in two months.
C3: Three people are available to do the programming.
```

4.2 Functional Specification

A functional specification is a black-box model of the proposed software system's behavior which consists of the system's interactions with other systems. The major goals in constructing functional specifications are to:

- define the external interfaces of the proposed system,
- check that the proposed system solves the customer's real problem, and
- check the feasibility of the proposed system.

The functional specification can be developed in two phases: (1) abstract functional specification which is a black-box description of each major subsystem to be built that describes the information content, and (2) concrete functional specification which describes the formats of the data crossing the boundaries of the system via transformations from concrete external inputs to abstract inputs, and abstract outputs into concrete external outputs.

In this approach, central modules are not decomposed into lower-level modules in the functional specification, as in many popular informal approaches. Instead, black-box specifications are partitioned by subsystems, interfaces, messages, and responses to show the structure of the system's functionality. The Spec language uses the *event model* to define the black-box behavior of proposed and external systems. The event model's four kinds of primitives (modules, messages, events, and alarms) form the semantic basis for Spec. Modules have no visible internal structure and can be used to model external systems such as users and peripheral hardware devices, as well as software components. Messages can be used to model user commands, system responses, and interactions between internal subsystems.

4.2.1 Abstract Functional Specification for the FFCS

For abstract functional specification, we consider only the information content of the messages crossing the boundaries of the proposed system, instead of the data formats of those messages. The first step is to identify external systems and major internal subsystems. In the FFCS problem, there is one proposed system, the `fish_farm_control_system`, and two external systems, the `operator` representing the human user and the `fish_tank` representing the external hardware system. The example is not large enough to separate the proposed system into multiple major internal subsystems.

Fig. 20 shows the top level interface for the FFCS. This corresponds to the context diagram in Fig. 1, but is simpler because individual data flows are not shown and because related terminators are aggregated into external systems (the sensors and valves are different functional aspects of the fish tank). This approach enables tractable descriptions of larger systems.

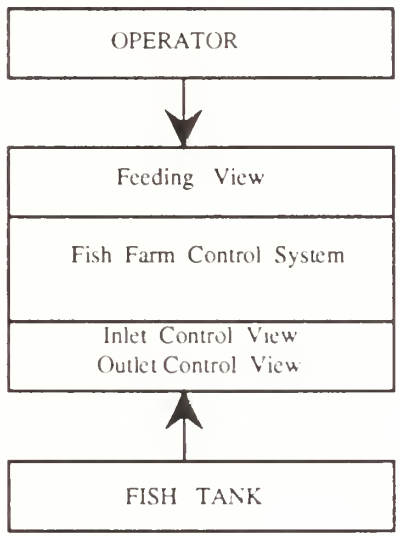


Fig. 20 Top Level Interface for the FFCS

The operator interface and the fish tank interface correspond to two different views of the proposed `fish_farm_control_system`. The interface views partition the set of stimuli and drive a divide and conquer strategy for determining an event list as shown in Fig. 21. The benefit of this is systematic separation of concerns, which is a major theme of the Spec method, are most apparent for complex systems much larger than the FFCS, which can have dozens or hundreds of separate interfaces.

```
operator view
  MESSAGE display_feeding_schedule  R1.1.1.1.1, R1.2.3.1
  MESSAGE update_feeding_schedule   R1.1.1.1.2, R1.2.3.1
  TEMPORAL feeding_time  R1.1.1.3, R1.2.3.1
fish tank view
  MESSAGE oxygen_level  R1.1.2.1, R1.2.2, R1.2.3.1
  MESSAGE ammonia_level R1.1.2.2, R1.2.2, R1.2.3.1
  MESSAGE water_level   R1.1.2.3, R1.2.2, R1.2.3.1
```

Fig. 21 Spec Event List

Both MESSAGE and TEMPORAL are Spec primitives. Messages represent arrival of input data for a module, while temporals are used to represent alarms triggering scheduled events. The requirements corresponding to each event are listed for traceability. The event list and the two actuators of the FFCS determine the decomposition of the functional specification shown in Fig. 22 and Fig. 23

```
MACHINE fish_farm_control_system
  INHERIT feeding_view
  INHERIT inlet_control_view
  INHERIT outlet_control_view
END
```

Fig. 22 Top Level Functional Specification

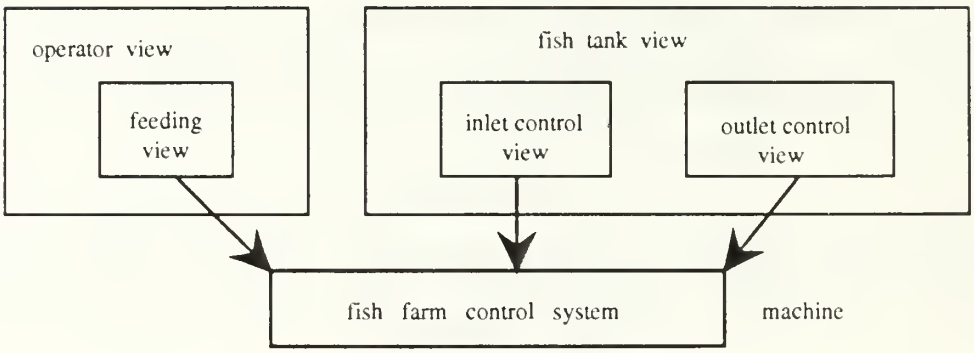


Fig. 23 Specification Inheritance for the FFCS

The behavioral specifications for the FFCS are given in Figs. 24 – 26.

```
MACHINE feeding_view
  INHERIT feeding_schedule_definitions
  IMPORT feeding feedings FROM feeding_schedule
  IMPORT second FROM time_unit

  STATE(schedule: feeding_schedule)
    INVARIANT true INITIALLY feedings(schedule) = { }

  MESSAGE display_feeding_schedule
    REPLY(s: feeding_schedule) WHERE schedule = s

  MESSAGE update_feeding_schedule(s: feeding_schedule)
    TRANSITION schedule = s

  TEMPORAL feeding_time
    WHERE time_to_start_feeding(schedule, TIME), DELAY <= (1 minute)
      CHOOSE(next_feeding: feeding SUCH THAT next_feeding
            IN feedings(schedule) & next_feeding.start = time_of_day(TIME) )
```

```
      SEND fish_feeder_on TO fish_tank WHERE DELAY = 0
      SEND fish_feeder_off TO fish_tank
        WHERE DELAY = (next_feeding.stop - next_feeding.start)
END
```

Fig. 24 Specification of Feeder Control

The specifications describe the required responses for each stimulus in the event list. Responses are described using either logical **WHERE** assertions or informal comments. Automated testing against the specification and proofs of code are possible if logical assertions are provided. Time delays can be constrained using postconditions, as illustrated in Fig. 24. The temporal event is defined by the condition `time_to_start_feeding`. The first **DELAY** statement says the temporal event must be recognized within one minute of the instant the condition becomes true. The second **DELAY** says the feeder must be turned on immediately when the temporal event is recognized, and the third **DELAY** says the feeder must be turned off when the scheduled feeding period has elapsed.

```
MACHINE inlet_control_view
  INHERIT sensor_state_view -- defines state variables oxygen, ammonia, and water
  INHERIT water_quality_definitions -- gives values for symbolic constants

  MESSAGE oxygen_level(level: concentration)
    WHEN level <= minimum_oxygen_level
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_open
        WHERE vs = #open, DELAY <= (1 second)
    WHEN all_levels_ok
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_closed
        WHERE vs = #closed, DELAY <= (1 second)
    OTHERWISE  -- Do nothing.

  MESSAGE ammonia_level(level: concentration)
    WHEN level >= maximum_ammonia_level
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_open
        WHERE vs = #open, DELAY <= (1 second)
    WHEN all_levels_ok
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_closed
        WHERE vs = #closed, DELAY <= (1 second)
    OTHERWISE  -- Do nothing.

  MESSAGE water_level(level: distance)
    WHEN level <= minimum_water_level
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_open
        WHERE vs = #open, DELAY <= (1 second)
    WHEN all_levels_ok
      SEND set_inlet_valve(vs: valve_setting) TO fish_tank -- inlet_valve_closed
        WHERE vs = #closed, DELAY <= (1 second)
    OTHERWISE  -- Do nothing.

  CONCEPT all_levels_ok: boolean
    WHERE all_levels_ok <=> oxygen >= minimum_oxygen_level &
```

```
                        water >= minimum_water_level &
                        ammonia <= maximum_ammonia_level
END
```

Fig. 25 Specification of Inlet Control

```
FUNCTION outlet_control_view
  MESSAGE water_level(level: distance)
    WHEN level >= maximum_water_level
      SEND set_outlet_valve(vs: valve_setting) TO fish_tank -- outlet_valve_open
        WHERE vs = #open, DELAY <= (1 second)
    WHEN level <= minimum_water_level
      SEND set_outlet_valve(vs: valve_setting) TO fish_tank -- outlet_valve_closed
        WHERE vs = #closed, DELAY <= (1 second)
    OTHERWISE  -- do nothing.
END
```

Fig. 26 Specification of Outlet Control

4.2.2 Concrete Functional Specification for the FFCS

The concrete functional specification specifies detailed formats for external inputs and outputs, including error messages, and define error correcting protocols. It defines the behavior of the software encapsulations for each external system, such as the operator and the fish tank. These encapsulations transform the abstract data types that appear in the abstract functional specification into the actual data representations that are passed physically to the external systems, and vice versa.

4.3 Architectural Design

The Spec approach to software development decomposes the proposed system into a hierarchy of small independent modules in an architectural design activity that follows functional specification. For the FFCS each message and temporal is simple enough to be implemented as a single unit, so that no additional decompositions are needed. The architectural design activity must also determine programming language representations for module interfaces, such as Ada package specifications. The last step can be automated.

5 Conclusion

Although the development of a real-time system typically requires a broad view at the level of system engineering, software engineers should be involved in the initial development of the requirements for the whole system. The requirements for such system must be satisfied with a combination of computer hardware, software, and special devices, and their formulation involves many trade-offs, such as the choice of what should be done in

software and what should be done in hardware. The process of developing software for such systems is complex and expensive. We have presented three methods that cover a wide spectrum of systematic approaches to real-time software development. This section draws distinctions between the three methods and summarizes the strengths as well as the weaknesses of each method.

5.1 Formality and Potential for Automation

Among the three methods, Structured Analysis represents the informal end of the spectrum, the Spec method represents the formal end, and the CAPS approach is semi-formal. All three methods can be considered systematic – the development process is divided into a sequence of well-defined steps, each of which has clearly defined results. It is only through a systematic approach that we can control large development efforts and produce high quality software and control its cost.

A systematic process alone, however, does not guarantee success, especially in large-scale software development activities, where precise communication and a high degree of automation are absolutely necessary. Precise communication is critical because in large projects a group of people must arrive at a consistent understanding of the proposed system to successfully build it. Automation is important at all stages of software development to improve accuracy and productivity. The degree of formality determines the amount of automation that is feasible. In a completely formal approach to software development, the process is represented, carried out, and evaluated by mathematical laws. This enables software tools to use mechanized reasoning to carry out parts of the process.

Structured Analysis relies on informal modeling tools such as dataflow diagrams, entity-relationship diagrams, and state transition diagrams. The method is simple and encourages modular design. However, it relies on human intelligence to provide precise interpretations for the documents, to produce explanations, to answer questions about the proposed design, or to produce implementations.

The CAPS approach, like Structured Analysis, is simple, encourages modular design, and uses graphics to represent system decompositions. The graphics are augmented with control constraints that can be used to expeditiously make small adjustments to the behavior of the prototype, without modifying the subsystems identified in a decomposition, which is useful in prototyping and exploratory design. The underlying semantic model is richer, formally defined, and more specific than the Structured Analysis models. The semantics of data streams includes implicit buffering, with a choice of two buffering disciplines determined by the control constraints. Consequences of this are that synchronization of data from different sources is automatic and that no special symbols for data stores are needed. The CAPS model is sufficiently formal to enable a variety of automated synthesis and analysis procedures, including real-time scheduling.

Spec is a formal specification language that is based on logic and spans the entire software development process. The Spec method supports development of very large systems that must be reliable. The Spec language is designed to localize information and to limit the size of the text that must be examined to understand each aspect of system behavior, so that complex systems can be kept intellectually manageable. The

logic is sufficiently expressive to capture all aspects of system behavior, and the model is sufficiently formal to support just about any computer-aided design process.

5.2 Automated Tool Support

There are commercial software tools such as Software Through Pictures to support Structured Analysis. However, automation is limited to graphical and text editing, structural consistency checking, and on-line manual configuration control. The method does not have automated support for analyzing the feasibility of timing constraints. All programming and testing tasks are done manually.

In addition to providing execution support to demonstrate system behavior and to check the validity of the requirements, CAPS also offers tools which span the entire software development process. In addition, fully-automated design-database and software-base systems are being developed to support configuration control and software reusability.

For Spec, tools are currently being developed to support a highly automated implementation process, including tools for detecting errors in the design, computer-aided construction and checking of program proofs, and for automatically testing implementations relative to the Spec, with required manual effort only for examining test results for failure cases.

5.3 Real-Time Features

Traditional Structured Analysis approach does not provide an integrated treatment of the real-time aspect of software systems. It adds control flows and control processes to the dataflow diagrams and uses a state transition diagram to model the internal states of control processes. The Hatley-Pirbhai extension allows the specification of some timing requirements: the repetition rate of primitive external signals and the maximum response time of the processes defined using state transition diagrams. These facilities do not cover all of the real-time constraints that arise in practice, and there is no mechanism to check the feasibility of timing constraints.

The CAPS method allows a decomposition to be augmented with timing constraints. CAPS has a richer set of timing specification features than the Hatley-Pirbhai extension of Structured Analysis, and it provides tools to check feasibility and to realize the timing requirements.

In Spec, real-time constraints can be specified and attached to an event or a transaction. The underlying logic can be used together with the use of events as reference points in time to express just about any timing constraint. A library of pre-defined abstractions to simplify the specification of sophisticated real-time systems could be defined using Spec, but such a library is not yet developed.

5.4 Learning the Methods

Conceptually, it is fairly easy to learn the underlying mechanism of Structured Analysis in a short period of time and apply it to real problems. The method has been popular and is the current state-of-the-art practice in the software industry. Although systematic in principle, its practice remains an art, not engineering nor science. Proficiency in applying this method is usually obtained only through years of practical experience.

CAPS is a sophisticated system, with its own set of integrated tools and the Prototype System Description Language. Prototype System Description Language can be learned with effort comparable to learning Structured Analysis. It takes somewhat longer to learn how to make the most effective use of all the software tools in the CAPS system. However, once the method is mastered, the increase in productivity associated with using the method should outweigh the extra time spent in training.

Spec is based on second order predicate logic, and requires considerably more training in mathematics and computer science concepts than the other methods. However, use of this method should enable significant increases in the reliability of the systems produced.

5.5 Software Life Cycle Activities

Structured Analysis is mainly for requirements analysis and system specification. Since all the documentations are informal, it is rather difficult, if not impossible, to verify the requirements formally. Although guidelines have been developed to ensure that the user requirements are accurately modeled and that the system specifications are complete and internally consistent, there is no way to enforce them. Implementation and testing are purely laborious work in this approach. When modification to the system is required, all the documentations have to be examined and changed manually, program recoded, and testing redone. The cost, time, and error rate associated with this manual effort encourage developers not to keep the requirements documents consistent with an evolving implementation.

The computer aided prototyping approach provides an efficient and effective mechanism for requirements tracing. In addition, one major advantage of the CAPS approach is rapid formulation of appropriate, useful, and feasible system specifications and designs.

The Spec method requires more design effort than the informal methods. However, this extra effort is justified in situations where people enter and leave a large development team, so that all decisions must be formalized (made explicit and recorded) rather than informal (left implicit by relying on "common sense"). The effort is also justified by the increased precision made possible by various supporting tools. The major advantage of the formal approach is that it allows automated testing and verification of implementations against the specification. This makes it more feasible to ensure that the specifications really correspond to the code, so that they can be of real utility during system maintenance.

5.6 Concluding Remarks

Structured Analysis is very handy for developing small and simple systems. CAPS is especially useful when the initial requirements are not clear or the feasibility of the system is uncertain. The Spec approach is necessary when reliable large systems are needed. We believe that each method has its own merits and limitations. The user should make the choice of the methods based on the nature of the application.

References

V. Berzins, Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.

D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specifications*, Dorset House, New York, 1987.

K. B. Kenny and K. Lin, *Building Flexible Real-Time Systems Using the Flex Language* IEEE Computer, May 1991.

Luqi, *Computer Aided Software Prototyping*, IEEE Computer, September 1991.

Luqi, *Real-Time Constraints in a Rapid Prototyping Language*, Journal of Computer Languages, to appear.

Luqi and V. Berzins, *Rapidly Prototyping Real-Time Systems*, IEEE Software, September 1988.

A. Mok, *A Systematic Approach to the Design of Hard Real-Time Systems*, ONR Workshop on Foundations of Real-Time Computing Research Initiative, Falls Church, Virginia, 29-30 November 1988.

Software Through Pictures, *Reference Manual*, Interactive Development Environment, San Francisco, California, 1990.

J. Stankovic, *Misconceptions About Real-Time Computing*, IEEE Computer, October 1988.

E. Yourdon, *Modern Structured Analysis*, Yourdon Press, New Jersey, 1989.

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22304–6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Computer Science Department Code CS Naval Postgraduate School Monterey, CA 93943	1
Prof. Yuh-jeng Lee Code CS/Le Naval Postgraduate School Monterey, CA 93943	30